Metric Space-based Test-data Adequacy Criteria

MARTIN DAVIS AND ELAINE WEYUKER

Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA

Since software testing cannot ordinarily be expected to provide conclusive evidence that a program is correct, software engineers have had to be satisfied with the vague notion of a set of test data being* adequate *for a given program. In this paper a theoretical model is provided for the notion of adequacy. Adequacy criteria are seen as serving to distinguish a given program from a certain class of programs. In particular, notions of* distance *between programs are studied, and adequacy of a test set is taken to mean that the set successfully distinguishes the program being tested from all programs that are sufficiently near to it, and differ in input–output behaviour from the given program. Certain points, called* critical, *are identified which must occur in every adequate test set. Finally, lower bounds are obtained on the size of test sets which are minimally adequate, in the sense that they have no adequate proper subsets.

Received May 1986, revised December 1986

1. INTRODUCTION

A test-data *adequacy criterion* provides explicit rules for determining when it is appropriate to end the testing phase of software development. When defining new adequacy criteria, it is necessary also to offer a plausible theoretical justification for the criteria proposed. (For example, see Refs 1, 3, 5 and 10.) Assuming, as we do here, that the criteria are to be *program-based*, an adequacy criterion may be thought of as an approximation to the unattainable ideal, whereby a set of test data T would be regarded as adequate for a given program P if the input–output behaviour of P on T distinguishes P from all programs Q whose input–output behaviour is not identical to that of P. Of course, in principle there are many ways to effect such an approximation. A particularly attractive way to proceed is to associate with each program P a finite set $\Phi(P)$ of programs such that for each $Q \in \Phi(P)$, Q is not equivalent to P; we can think of $\Phi(P)$ as a finite approximation to the set of *all* programs Q such that Q is not equivalent to P. We can then define a set of test data T as Φ-*adequate* for P if

$$(\forall Q \in \Phi(P))\,(\exists t \in T)\,(P(t) \neq Q(t)).$$

That is, instead of distinguishing P from all inequivalent programs, T need only distinguish P from programs in $\Phi(P)$. This approach was followed in Ref. 3, where a number of results were obtained by, in effect, taking $\Phi(P)$ to be the class of all programs Q such that Q is inequivalent to P, and Q has program size less than that of P, with respect to a suitable notion of size. The choice for $\Phi(P)$ in Ref. 3 was, of course, somewhat ad hoc; there is no special reason to single out programs of size smaller than the program being tested, except that there are only finitely many such programs. In this paper we generalise our notion of adequacy and consider a family of adequacy notions all based on taking $\Phi(P)$ to be the set of programs Q inequivalent to P which are *near* P in some appropriate sense. We shall not restrict ourselves to any one notion of 'near', but will instead construe 'nearness' as referring to a measure of *distance* between programs assumed to satisfy the metric space axioms.

This research was supported by Office of Naval Research contract number N000014-85-K-0414 and by National Science Foundation grant number DCR-85-01614. The authors wish to acknowledge with thanks suggestions made by the referee. In particular, it was at the referee's suggestion that proofs of theorems were placed in an appendix.

A fundamental difficulty in the general approach considered above arises from the fact that $\Phi(P)$ may well contain a program Q, 'part' of which is equivalent to P. As we shall see, such programs Q can be used in diagonal constructions to show that, except for very special circumstances, no program can have a Φ-adequate test set. We deal with this problem by excluding from $\Phi(P)$ all programs Q in which P is *embedded*, in the sense defined below. (This is essentially the same formal solution to this problem used in Ref. 3.) Since the definition of *embedded* involves syntactic transformations of a program, we shall need to be very precise about our programming language.

Thus we shall begin by specifying our programming language. Next, we shall show how vulnerable adequacy notions are to diagonal constructions. We shall define the notion of a program being 'embedded' in another, indicating how this notion enables us to avoid the above difficulty. Finally, we shall turn to distance-based adequacy notions. Although the reader of Ref. 3 will note some similarities, we do not assume acquaintance with that paper. Also, we assume no previous knowledge of metric spaces.

2. THE PROGRAMMING LANGUAGE

We assume a finite number of identifiers whose range is the integers (positive, negative or zero) as well as a finite number of constants representing particular integers; it is assumed that numbers encountered as input or output values are represented by corresponding constants of the language. Arithmetic expressions are constructed using constants, identifiers, parentheses and the arithmetic operators $+$, $-$, $*$, $/$ as usual. An assignment statement has the form:

$$\text{VAR} \leftarrow \text{EXP}$$

where VAR is an identifier and EXP is an arithmetic expression. The null statement Λ is the empty string and represents a NO−OP. A predicate is to be of one of the forms:

$$B_1 = B_2, \quad B_1 \neq B_2, \quad B_1 < B_2, \quad B_1 \leqslant B_2,$$

where B_1 and B_2 may each be a constant or an identifier. We next define program body.

(1) Λ is a program body.

(2) An assignment statement is a program body.

(3) If *PRED* is a predicate and P and Q are program bodies, then

if *PRED* **then** P **else** Q **end**

is a program body.

(4) If *PRED* is a predicate and P is a program body, then

if *PRED* **then** P **end**

is a program body.

(5) If *PRED* is a predicate and P is a program body, then

while *PRED* **do** P **end**

is a program body.

(6) If P and Q are program bodies so is PQ.

If VAR is an identifier then

input VAR and **output** VAR

are called **input** and **output** statements, respectively. Finally, an input statement followed by a program body followed by an output statement is called a *program*, and the program body in question is called the *body* of the program. We shall often permit ourselves the 'abuse of language' of failing to distinguish between a program and its body. For example, we speak of the program Λ, meaning a program whose body is Λ.

3. Φ-ADEQUACY

We shall write $P(c) = b$ to mean that on input c, program P halts with output b and $P \equiv Q$ (P is *equivalent* to Q) to mean that P and Q halt on the same inputs and that $P(c) = Q(c)$ for all such inputs c. When we write $P = Q$, we mean that P and Q are syntactically identical. Now let Φ be a mapping that associates with each program P a finite set $\Phi(P)$ of programs inequivalent to P.

Definition

A set of inputs T is Φ-*adequate* for the program P if for every program $Q \in \Phi(P)$ there is a point $t \in T$ such that $P(t) \neq Q(t)$.

Thus we are taking the position that choosing an adequacy criterion amounts to selecting a particular mapping Φ, yielding a set of inequivalent programs from which P must be distinguished. Our first theorem shows that care must be taken in specifying $\Phi(P)$. The proofs of all our theorems will be found in the Appendix.

Theorem 1 (Reflexivity Theorem)

If for some c not in T, $\Phi(P)$ contains a program Q of the form:

input x
if $x = c$ **then** $y \leftarrow b$ **else** R **end**
output y

where $R \equiv P$ and $b \neq P(c)$, then T is not Φ-adequate for P.

The practical implications of this theorem are important. Of course, one desires finite test sets which do not exhaust the domain of the program being tested. Thus it is surely intended that for a test set T for a given program there will ordinarily be points belonging to the domain of the program that are not in T. However, most reasonable criteria for the set $\Phi(P)$ do not automatically exclude programs like the program Q in the statement of the Reflexivity Theorem. Therefore, unless we can find a way to exclude such programs, this theorem implies that no program can be adequately tested short of exhaustive testing. Our method of seeking to banish such programs will make use of the following seven reduction rules.

(1) Replace some assignment statement by Λ.

(2) Replace an **if** statement: **if** *PRED* **then** P **else** Q **end** by P.

(3) Replace an **if** statement: **if** *PRED* **then** P **else** Q **end** by Q.

(4) Replace an **if** statement: **if** *PRED* **then** P **end** by P.

(5) Replace an **if** statement: **if** *PRED* **then** P **end** by Λ.

(6) Replace a **while** statement: **while** *PRED* **do** P **end** by P.

(7) Replace a **while** statement: **while** *PRED* **do** P **end** by Λ.

We say that a program M *reduces to* N if the program N can be obtained from M by 0 or more applications of these reduction rules, and that M is *embedded* in N if N reduces to some program which is equivalent to M. Clearly, if M is equivalent to N, M is embedded in N, and N is embedded in M.

This way of thinking about test-data adequacy criteria helps elucidate a fundamental weakness of using statement or branch coverage as such a measure. A Φ-adequate criterion requires that test data be included which distinguish a given program from a fixed set of other inequivalent programs. Viewed another way, such a criterion requires test data which guarantee that certain predetermined errors are not present.[7] Statement and branch testing, in contrast, do not seek to distinguish a given program from other programs (or to detect particular errors). They merely require that various parts of the program be executed. The connection between this type of code exercising and the location of errors is at best indirect. However, since it is clear that statement and branch coverage represent necessary conditions for test-data adequacy (if a portion of a program has never been executed, any incorrect code could be within that subprogram), it is important to observe that there are Φ-adequate criteria which imply branch adequacy (which, in turn, implies statement adequacy).

Theorem 2

Let $\Phi(P)$ consist of all programs inequivalent to P to which P can be reduced using rules 2, 3, 4, 5, 6 and 7. If T distinguishes P from $\Phi(P)$, then T results in each branch of P being traversed.

Noting that what makes the program Q in the statement of the Reflexivity Theorem work is that P is embedded in it, we are led to call a mapping Φ *protected for program* P, if $\Phi(P)$ contains no program in which P is embedded. Thus, if Φ is protected for P, *the Reflexivity theorem is no barrier to the existence of a non-exhaustive adequate test set for* P. Note also that, since P is obviously embedded in any program Q such that $Q \equiv P$, if Φ is protected for P, then $\Phi(P)$ contains no program equivalent to P. Thus, by confining our attention to mappings that are protected for P, we can omit from our definition the requirement that the set $\Phi(P)$ should contain only programs inequivalent to P. If Φ is protected for all programs P, then we say that Φ is *totally protected.*

Theorem 3

If the mapping Φ is protected for program P, there is a finite set T which is Φ-adequate for P.

Thus, in the most general case, we view a program as being adequately tested if the test data have distinguished it from all programs in a predesignated set of alternative programs. This set can be viewed as being determined by a mapping Φ. We have seen that if restrictions are not placed on Φ to prevent embedded programs, one cannot expect to be able to Φ-adequately test an arbitrary program using less than exhaustive testing.

In the next section we examine a particular set of mappings which, we believe, yield an interesting and intuitively reasonable set of alternative programs.

4. METRIC SPACE ADEQUACY

A program is tested in order to find *errors*. Each such error represents a way in which the program being tested fails to perform as desired or, to put it another way, in which it is not the program one wishes it to be. Using the language of Φ-adequacy, we may say that the hope is that the desired correct program Q is contained in the set $\Phi(P)$, where P is the program being tested. When the error to be found is a 'minor' one, one will expect that P and Q will not be very different syntactically. Usually, when one speaks of a 'major' error, one can expect that P will differ more substantially from Q. These considerations suggest that we consider a hierarchy of adequacy notions in which $\Phi(P)$ consists of programs which are syntactically different from P to a greater and greater extent. This in turn requires us to be able to speak quantitatively of the extent to which programs differ or, as we shall say, of the *distance* between programs.

In considering what properties one would wish a notion of distance between programs to possess, we have been guided by the study of metric spaces in set-theoretic topology. Thus we define a distance function between programs to simply be one which satisfies the traditional axioms for metric spaces. More specifically, we proceed as follows:

By a *distance function* we shall mean a real valued function ρ on pairs of programs, such that for all programs P, Q and R:

(1) $\rho(P,Q) \geqslant 0$;
(2) $\rho(P,Q) = 0$ if and only if $P = Q$;
(3) $\rho(P,Q) = \rho(Q,P)$;
(4) $\rho(P,R) \leqslant \rho(P,Q)+\rho(Q,R)$.

These are the well-known axioms for a metric space; in particular (4) is called the *triangle inequality*. If for each P and each $d > 0$ there are only finitely many programs Q such that $\rho(P,Q) \leqslant d$, then ρ is called a *finite* distance function; if $\rho(P,Q)$ is always an integer, ρ is called a *discrete* distance function.

We shall not use any one particular distance function, but rather try to state our results in as general a form as possible. One example of a finite discrete distance function can easily be specified by using the reduction rules which were introduced in order to define the notion of one program being embedded in another. That is, we can let $\rho(P,Q)$ be the least integer n such that there is a sequence:

$$P = P_1, P_2, \ldots, P_n = Q,$$

where for $i = 1,2,\ldots,n-1$, either P_i reduces to P_{i+1} or vice versa, using one application of the rules 1 to 7 above. However, although this function ρ does satisfy our formal definitions, it cannot really be regarded as satisfactory. For example, by reduction rule 2, the program $x \leftarrow 0$ is at a distance 1 from the program

if *PRED* **then** $x \leftarrow 0$ **else** Q **end**

no matter how complicated Q may be. Later we shall see how to construct a more reasonable distance function using a grammar for our programming language.

An example of a function that does *not* satisfy the definition of a distance function is $f(P,Q)$, defined to be the number of points c such that $P(c) \neq Q(c)$ if this number is less than some constant K, and defined to be K otherwise. This definition fails to satisfy axiom 2, since for any pair of (semantically) equivalent programs P and Q, $f(P,Q) = 0$, even though P and Q may be substantially different syntactically. Note in addition that for a given P and d there may be infinitely many programs Q such that $f(P,Q) \leqslant d$.

A useful heuristic guide is that however distance is measured, the function $\rho(P,\Lambda)$ should serve as a program complexity measure.[11] Of course, by the triangle inequality,

$$\rho(P,Q) \leqslant \rho(P,\Lambda)+\rho(Q,\Lambda).$$

Now, let ρ be a given *finite* distance function. Then, for each $d \geqslant 0$, we let $\Phi_d(P)$ be the set of all programs Q such that:

(1) $\rho(P,Q) \leqslant d$;
(2) P is not embedded in Q.

The definition implies that each Φ_d is totally protected. In a context where a fixed distance function may be assumed, we shall say that a set T is *d-adequate*, meaning that it is Φ_d-adequate. Obviously if T is d-adequate and $d > e$, then T is e-adequate.

It is interesting to compare the present approach to adequacy to that arising in work on mutation analysis.[1,2,4] Mutation analysis is a program-based testing technique in which a primary underlying assumption is that 'competent programmers' produce programs which are 'close' to correct. Therefore, it is argued, if the program being tested is distinguished from all inequivalent programs which are 'close' to it, then the tester can conclude that the program has been well tested. Thus a mutation system would first generate a number of programs, each of which differs from the original by one simple syntactic change (similar to our reductions). These modified programs are known as *mutants*. In order to distinguish the original program from each of the inequivalent mutants, test data must be supplied which cause the original and mutated programs to produce different outputs. Thus *mutation adequacy* is essentially 1-adequacy, for the particular distance function just defined using our reduction rules.

Much of the appeal of the mutation concept comes from the intuition provided by the 'coupling effect',[4] which states that 'test data that distinguishes all programs differing from the correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors'. Given this intuition, it seems reasonable to restrict attention to test data which distinguish a program from programs that are 'near' it. The idea of d-adequacy generalises the mutation notion in two significant ways: it permits one to abstract from any one particular way of measuring the distance between programs, and it enables one to study sequences of adequacy notions of increasing scope. More will be said about this below.

5. A GRAMMAR-BASED DISTANCE FUNCTION

We begin by specifying a context-free (i.e. BNF) grammar for our programming language. The first two productions listed are left indefinite to correspond to our assumption of arbitrary finite lists of constants and identifiers.

$\langle constant\rangle \to c_1|c_2|\ldots|c_m$
$\langle identifier\rangle \to v_1|v_2|\ldots|v_n$
$\langle operator\rangle \to +|-|*|/$
$\langle relation\rangle \to =|\neq|<|\leqslant$
$\langle expression\rangle \to c_1|c_2|\ldots|c_m$
$\langle expression\rangle \to v_1|v_2|\ldots|v_n$
$\langle expression\rangle \to \langle expression\rangle\langle operator\rangle\langle expression\rangle$
$\langle expression\rangle \to (\langle expression\rangle)$
$\langle predicate\rangle \to \langle constant\rangle\langle relation\rangle\langle constant\rangle$
$\langle predicate\rangle \to \langle identifier\rangle\langle relation\rangle\langle constant\rangle$
$\langle predicate\rangle \to \langle constant\rangle\langle relation\rangle\langle identifier\rangle$
$\langle predicate\rangle \to \langle identifier\rangle\langle relation\rangle\langle identifier\rangle$
$\langle program\text{-}body\rangle \to \Lambda$
$\langle program\text{-}body\rangle \to \langle identifier\rangle \leftarrow \langle expression\rangle$
$\langle program\text{-}body\rangle \to$ **if** $\langle predicate\rangle$ **then** $\langle program\text{-}body\rangle$ **else** $\langle program\text{-}body\rangle$ **end**
$\langle program\text{-}body\rangle \to$ **if** $\langle predicate\rangle$ **then** $\langle program\text{-}body\rangle$ **end**
$\langle program\text{-}body\rangle \to$ **while** $\langle predicate\rangle$ **do** $\langle program\text{-}body\rangle$ **end**
$\langle program\text{-}body\rangle \to \langle program\text{-}body\rangle\langle program\text{-}body\rangle$
$\langle program\rangle \to$ **input** $\langle identifier\rangle\langle program\text{-}body\rangle$ **output** $\langle identifier\rangle$

In order to use this grammar to define a distance, we speak of a *transition sequence* from program P to program Q, meaning a sequence of expressions $P_1, \ldots, P_k$ such that $P_1 = P$ and $P_k = Q$ and for each $i = 1, \ldots, k-1$ either P_{i+1} is obtainable from P_i (referred to as a *forward step*) or P_i is obtainable from P_{i+1} (a *backward step*) using one of the productions of the grammar. Next we define the *length* of a transition sequence to be the maximum of the number of forward steps and the number of backward steps in the given transition sequence. Now we can define $\rho(P, Q)$ as the smallest length of a transition sequence from P to Q. It is very easy to see that this function ρ is indeed a distance function, and is in fact both finite and discrete. However, ρ as thus defined is not really very satisfactory, because there are many examples of pairs of programs which intuitively should be very close, but which are a rather large distance apart as measured by ρ as thus defined.

To deal with this problem, we add productions to our grammar intended to provide 'short cuts', that is to bring closer programs whose distance apart is greater than seems intuitively appropriate. For this purpose, we add to our grammar a special non-terminal $\langle|\rangle$. Although we shall add productions involving $\langle|\rangle$, the symbol $\langle|\rangle$ will never appear on the right-hand side of a production. Hence these productions will not change our language at all. However, because transition sequences are permitted to contain backward steps, these productions can and will decrease the distance between programs. The first additional production is

$\langle|\rangle \to \Lambda$

Thus by inserting a backward step into a transition sequence, the symbol $\langle|\rangle$ may be inserted anywhere. Desired 'short cuts' can then be obtained simply by inserting appropriate productions with $\langle|\rangle$ on the left. In particular, we add the additional productions:

$\langle|\rangle \to \langle operator\rangle\langle expression\rangle$
$\langle|\rangle \to \langle expression\rangle\langle operator\rangle$
$\langle|\rangle \to)$
$\langle|\rangle \to ($
$\langle|\rangle \to \langle expression\rangle\langle operator\rangle)$
$\langle|\rangle \to)\langle operator\rangle\langle expression\rangle$

Now we define ρ exactly as above, but permit the use of these additional productions in constructing transition sequences.

To help in understanding the effect of these short cuts, let us consider what types of syntactic change cause programs P and Q to be 'very close' to each other, i.e. distance 1, using this notion of distance. If the only difference between P and Q is that the relation symbol in a predicate is different, or the operator in an expression is different, these programs would be distance 1 apart. Similarly if a new program is obtained by changing a single identifier or constant at exactly one place, the distance between these programs would be 1. Likewise if the new program is obtained by replacing a constant by an identifier, or vice versa, at exactly one place, the resulting program is at a distance 1 from the original program.

On the other hand, adding an assignment statement to a program requires, at a minimum, the following transition sequences:

Λ, $\langle program\text{-}body\rangle$, $\langle identifier\rangle \leftarrow \langle expression\rangle$, $v_i \leftarrow \langle expression\rangle$, $v_i \leftarrow v_j$

The distance between a given program and one with this simplest type of assignment statement added is 3 (the transition sequence involves 1 backward step and 3 forward steps).

Also distance 3 apart are programs P and Q, which differ only in one statement in which P contains the statement

$$y \leftarrow x$$

while Q contains the statement

$$y \leftarrow x+1$$

In contrast the program R which is identical to P except that the statement

$$y \leftarrow x$$

is replaced by

$$y \leftarrow 2*z$$

would be of distance 4 from P, as demonstrated by the transition sequence

$y \leftarrow x$, $y \leftarrow \langle expression\rangle$,
$y \leftarrow \langle expression\rangle\langle operator\rangle\langle expression\rangle$,
$y \leftarrow 2\langle operator\rangle\langle expression\rangle$,
$y \leftarrow 2*\langle expression\rangle$, $y \leftarrow 2*z$

which contains 1 backward and 4 forward steps.

Inserting (or deleting) a **WHILE** statement with a non-null body requires a transition sequence containing a minimum of 8 forward (backward) steps, as does an **IF THEN** statement with a non-null program body. Inserting an **IF THEN ELSE** statement with neither program body null requires a minimum of 11 forward steps. Of course, as the number of statements and degree

of complication of the program bodies grow, so does the number of steps needed to insert or delete code, and hence the distance between programs.

We will sometimes speak of the distance between pairs of statements or parts of statements to mean the obvious application of the distance function to the desired program parts.

6. A REPRESENTATION THEOREM

Let us call a mapping Φ *symmetric* if it satisfies the following condition.

If P is not embedded in Q and Q is not embedded in P, $Q \in \Phi(P)$ if and only if $P \in \Phi(Q)$

Then it is clear that the mappings Φ_d defined by a distance function are symmetric. In fact we can easily prove:

Theorem 4 (*Representation Theorem*)

Let the mapping Φ be symmetric. Then there exists a distance function ρ such that a set T is Φ-adequate for a program P if and only if T is 1-adequate for P with respect to ρ.

Critical points

Generalizing from Ref. 3, we define as follows.

Definition

A point c is called Φ-*critical* for a program P if there exists a program $Q \in \Phi(P)$ such that $P(t) = Q(t)$ for all $t \neq c$, but $P(c) \neq Q(c)$.

Theorem 5

Let T be Φ-adequate for program P and let c be Φ-critical for P. Then $c \in T$.

In what follows, we assume that some fixed specific finite-distance function has been chosen, and we write *d-critical* for Φ_d-critical. Then we have:

Theorem 6

Let c be d-critical for P and let $d < e$. Then c is e-critical for P.

We now assume temporarily that the distance function referred to is the one defined above in terms of a grammar.

Consider a program P:

if $x \leqslant c$ **then** Q **else** R **end**

where $Q(c) \neq R(c)$. Then c is 1-critical for P as demonstrated by the program:

if $x < c$ **then** Q **else** R **end**

Similarly $b = c+1$ is 1-critical for P as evidenced by the program:

if $x \leqslant b$ **then** Q **else** R **end**

This type of 'boundary' point is the same type that was found to be critical in Ref. 3.

The question is, then: does considering increasingly distant programs as possible alternatives to the program being tested increase the number of points which *must* be included in any d-adequate test set for the program? The surprising answer is yes, but not until d becomes quite large, and only for certain kinds of programs.

Let $f(x)$ be an arithmetic expression containing the one identifier 'x'. The program $y \leftarrow f(x)$ will generally have no d-critical point, no matter how large d may be. In general, points of 'discontinuity', i.e. points on the boundary between regions in which essentially different computations are performed (like b and c in the above example), turn out to be 1-critical. Indeed, it is precisely because the program $y \leftarrow f(x)$ does not have any such discontinuity that it has no d-critical point. Now consider a program which computes $g(x)$ on some finite contiguous portion D_1 of the domain (where D_1 does not consist of a single point and where $g(x)$ is an arithmetic expression) but which computes $f(x)$ on D_2, the set of all points of the domain not in D_1. As in the above example, such a program will normally have the usual endpoints of D_1 and D_2 as 1-critical points. However, each of the points of D_1 will also be d-critical for sufficiently large d. A simple example will help to clarify the intuition and give the reader a feeling for the size of d which would be required.

Consider the program P below, in which c_1, c_2 are constants such that $c_1 \leqslant c_2$:

$y \leftarrow f(x)$
if $x \geqslant c_1$ **then if** $x \leqslant c_2$ **then** $y \leftarrow g(x)$ **end end**

The points $c_1 - 1$, c_1, c_2, $c_2 + 1$ would all normally be 1-critical points.

Next, consider the program Q:

$y \leftarrow f(x)$
if $x \geqslant c_1$ **then if** $x \leqslant c_2$ **then** $y \leftarrow h(x)$ **end end**
if $x = c_1$ **then** $y \leftarrow g(c_1)$ **end**
if $x = c_1 + 1$ **then** $y \leftarrow g(c_1 + 1)$ **end**
⋮
if $x = c_2$ **then** $y \leftarrow g(c_2)$ **end**

where the block of **if** $x = c$ **then** $y \leftarrow g(c)$ statements contains one for each element of D_1 except the point c_3. Then, since P is not embedded in Q provided $h(x) \neq g(x)$ for all $x \in D_1$, it follows that c_3 is d-critical for P, where d is the distance between P and Q. This distance will depend on such factors as the size of D_1 and the arithmetic expressions g and h. Assuming that the distance between g and h is 1 (for example that they differ by a single constant or arithmetic operator), then, since the addition of each of the **if** statements requires 8 forward steps, $\rho(P, Q) = 1 + 8k$ where D_1 contains $k+1$ points, and hence each of the points in D_1 would be $(1+8k)$-critical.

The implications of this example are quite interesting. A critical point *must* be included in any adequate test set. Intuitively, these are points at which a programmer is apt to make a mistake and therefore they should always be checked. Off-by-one errors are a classical example of a programming blunder. The programmer includes one too few or one too many elements in a subdomain. It is interesting to notice that the points selected by our theory as necessary to test coincide with the points which pragmatic experience has taught us should be included in every test set. We consider this a confirmation of the reasonableness of our theory.

In contrast to 1-critical points, the type of error represented by the last example seems much less likely, and this is reflected in the distance from P to Q, and the

unnatural type of program which must be constructed in order to demonstrate that such a point is d-critical for suitable d.

We conclude this section with two slightly more substantial examples. Consider the following program, which is intended to compute the integer part of the square root of a number n, provided $0 \leqslant n \leqslant 100$. 'Error' indicates that the program prints some error message (technically an integer in our language) and halts.

```
input n
if n < 0 then error
        else if 100 < n then error
                        else i ← 0
                             j ← 1
                             while j ⩽ n do
                                   i ← i+1
                                   j ← (i+1)*(i+1)
                             end
                        end
        end
output i
```

The 1-critical points for this program include -1, 0, 1, 100 and 101. -1, 0, 100 and 101 arise from the bounds on n. 0 and 1 are also critical points which arise from the statement which initialises j (namely $j \leftarrow 1$). If instead j were initialised to 0, the square root of 0 would be computed to be 1, rather than 0. No other input would be affected, and hence 0 is a 1-critical point. Similarly, if j were initialised to 2, the square root of 1 would be determined to be 0, and hence 1 is 1-critical. 0 can also be seen to be 1-critical by changing the initialisation of i from '$i \leftarrow 0$' to '$i \leftarrow -1$'. For input 0, the output would then be computed to be -1. All other outputs would remain unchanged. As explained above, if d is chosen to be sufficiently large, each of the points in the interval [0, 100] becomes d-critical for this program.

As mentioned above, some programs, particularly those that compute the same function on every element of the domain, will generally have no d-critical points. For example, if the restriction on the upper bound on n were removed, 100 and 101 would no longer be 1-critical, and the points in the interval [2, 100] would no longer be d-critical. If the restriction that $n \geqslant 0$ were removed, -1 would no longer be a 1-critical point, but 0 and 1 would remain 1-critical points, as explained above. In this case the program computes 0 for $n \leqslant 0$ and $\lfloor \sqrt{n} \rfloor$ for $n \geqslant 1$. 0 and 1 are 'points of discontinuity' between the two functions, and hence are 1-critical points. Notice that changing the predicate '$j \leqslant n$' to '$j < n$' yields no critical point, since *many* inputs would now have an incorrectly computed square root. On the other hand, since the given square root program and the version with the incorrect predicate are distance 1 apart, some test case must be included which exposes this boundary error in order for the program to be deemed adequately tested.

Our final example is intended to help elucidate the notion of a critical point and the type of test sets needed to satisfy our criterion. The program is supposed to categorise triples of integer inputs, when interpreted as the lengths of the sides of a triangle, and is similar to programs in Refs 4, 8 and 9. A specification for such a program is also presented in Ref. 6 as an exercise to check one's skills in designing test sets. The program prints a message indicating whether the inputs represent a scalene, isosceles or equilateral triangle, or do not represent the sides of a triangle. For simplicity, in exhibiting this program we permit ourselves some inessential extensions of our programming language. Thus we show outputs as strings with appropriate semantic content (error, scalene, etc.) where strict conformity to our programming language would require integers. Also, we permit input of tuples of integers (as is the case in the closely related language used in Ref. 12). Finally, we use simple arithmetic expressions in the predicates (where strictly speaking we are only allowed constants and variables).

```
input A, B, C
if A ⩾ B+C
then error
else if B ⩾ A+C
     then error
     else if C ⩾ A+B
          then error
          else if A ≠ B
               then if A ≠ C                    i
                    then if B ≠ C
                         then scalene
                         else isosceles end     l
                    else isosceles end
               else if A ≠ C
                    then isosceles
                    else equilateral end
               end
          end
     end
end
```

This example contains no d-critical point for any value of d. This is consistent with one's intuition that there is no particular point (in this case triple) which *must* be included in every test set. On the other hand, however, the criterion does require that a representative from each of several classes be included. For example, in order to distinguish the given program from the program in which the first predicate '$A \geqslant B+C$' is replaced by the predicate '$A > B+C$', it is necessary to include a test case for which $A = B+C$. Similarly, from the second and third predicates, test cases with $B = A+C$ and $C = A+B$ must be included. To distinguish the given program from one in which the first predicate is replaced by '$A = B+C$', test data must be included for which $A > B+C$. Similarly, from the second and third predicates, we must include test cases with $B > A+C$ and $C > A+B$. These six test-case types represent all the ways that a triple may fail to satisfy the triangle inequality. If one continues this analysis considering programs which are distance 1 from the original, it is easy to demonstrate that the tester would also be required to include test cases representing valid scalene, isosceles and equilateral triangles (including all permutations of the two equal sides of an isosceles triangle, and all permutations of the relative sizes of a scalene triangle). If one considers Myers' list of appropriate test cases for such a triangle classification program,[6] we find that representatives of 8 of his 13 'likely errors' must be included in any 1-adequate test set. If, however, the distance is increased, the three additional 'likely errors' involving negative and 0 inputs must be included. They are deemed less likely than the other errors by our criterion in the sense that they represent programs which are a greater distance from the original than the other cases. If one examines this more

closely, this can be seen to be intuitively reasonable, since having a negative-length side or a side of length 0 is simply a special case of the failure of the triangle inequality. For example, if $A \leqslant 0$ either $C \geqslant B+A$ or $B \geqslant C+A$, and hence the triangle inequality is not satisfied. Thus our theory would require that test data be included which would expose 11 of 13 of Myers' 'likely errors'. Myers lists two other 'likely errors' which should be tested for. The first is a non-integer-type input. Since our language contains only integers, this is not a possible error in our theory. The other error type is an incorrect number of inputs. What happens when a program has been provided with too few inputs depends on just how uninitialised variables are treated. Since we have not chosen to deal with this matter in formulating our programming language, we cannot expect to handle errors of this kind.

7. MINIMALLY ADEQUATE TEST SETS

In general, we say that a set T is *minimally* Φ*-adequate* for a program P if:

(1) T is Φ-adequate for P;

(2) if $S \subseteq T$ and S is Φ-adequate for P, then $S = T$.

We have seen that for a program P with protected mapping Φ, there is a finite Φ-adequate test set. By successively deleting points from such a Φ-adequate test set, eventually we must arrive at a minimally Φ-adequate test set. Of course, we can now speak of a set T being *minimally d-adequate* for a program P. Let us call a distance function ρ *stepwise* if for every program Q of the form

if $x = c$ **then** $y \leftarrow b$ **else** P **end**

we have $\rho(P, Q) = 1$. Our grammar-based distance function is not stepwise in this sense. However, it is not difficult to see how it could be transformed into a stepwise distance function. Begin by adding to the grammar the 'short cut' productions:

$\langle | \rangle \rightarrow$ **if** $v_i = c_j$ **then** $v_k \leftarrow c_l$ **else** $1 \leqslant i, k \leqslant n;$
$1 \leqslant j, l \leqslant m.$

$\langle | \rangle \rightarrow$ **end**

Then for P and Q as above we have $\rho(P, Q) = 2$. Then a change of scale (replacing each distance d by $\lceil d/2 \rceil$) suffices to make ρ into a discrete, stepwise distance function. Of course, letting R be the program **if** $x > c$ **then** $y \leftarrow b$ **else** P **end**, $\rho(P, R)$ is a good deal larger than 1, although intuitively Q and R seem equally distant from P. Presumably, by adjoining still more 'short-cut' productions we could make our distance function behave in a manner more in accord with intuition. However, since the notion of ρ being stepwise occurs as a hypothesis in only one of our theorems, we do not dwell on the matter.

Now, let ρ be a discrete, finite, stepwise distance function, and let

$$T = \{c_1, c_2, \ldots, c_k\}$$

be d-minimally adequate for program P. For $1 \leqslant i \leqslant k$, let

$$T_i = T - \{c_i\}.$$

Then, T_i cannot be d-adequate for P. Hence, for each such i there is a program P_i such that

(1) $\rho(P_i, P) \leqslant d$,

(2) P is not embedded in P_i,

(3) $(\forall t \in T_i)(P(t) = P_i(t))$,

(4) $P_i(c_i) \neq P(c_i)$.

Then let Q_i be the program:

if $x = c_i$, **then** $y \leftarrow b$ **else** P_i **end**

where $b = P(c_i)$. This leads to our final result.

Theorem 7 (*Dichotomy Theorem*)

Let ρ be a finite discrete stepwise distance function with respect to which $T = \{c_1, c_2, \ldots, c_k\}$ is a minimally d-adequate test set for a program P which outputs at least 2 distinct values. Let P_i and Q_i be as above for $i = 1, 2, \ldots, k$. Then for each i, either c_i is a d-critical point for P or $\rho(P, Q_i) = d+1$.

Thus the Dichotomy Theorem implies that the number of non-d-critical points in a minimally d-adequate test set can be no larger than the number of distinct programs at a distance $d+1$ from P.

8. CONCLUSIONS

We have proposed a theoretical framework for studying test-data adequacy criteria in which a test set T is regarded as adequate for a program P, provided T distinguishes P from every inequivalent program in a pre-designated finite set of alternative programs.

What varies with the criteria is the way in which the set of alternative programs is selected. This is a natural way of characterizing adequacy criteria. Ideally, a test set should be able to distinguish a program from all inequivalent programs. Since there are, in general, infinitely many programs which are inequivalent to a given program P, this is clearly impossible. Therefore any pragmatically usable adequacy criterion provides a rule for selecting a finite subset of the inequivalent programs from which P must be distinguished.

We next proposed a hierarchy of adequacy notions, based on the 'distance' between two programs. We defined a highly 'sensitive' distance function and used it to investigate various specific examples in the light of our definitions. Associated with a given program, we were able to characterize certain points which we called *critical*. These points must be present in every adequate test set. Finally, by studying *minimally* adequate test sets, we obtained a lower bound on the number of *non-critical* points that must be present in an adequate test set.

We believe that there is much scope for further investigations along these lines. Other distance functions can be defined, and the behaviour of the class of critical points as distance functions are changed can be studied. The question should be investigated of whether there exist intuitively plausible adequacy notions that *cannot* be regarded as equivalent to d-adequacy with respect to some distance function. These investigations should also lead to helpful insights for strengthening the systems of axioms for adequacy studied in Ref. 12.

REFERENCES

1. T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, Theoretical and empirical studies on using program mutation to test the functional correctness of programs. *Proc. 7th Annual Symp. on Principles of Programming Languages, Las Vegas*, pp. 220–233 (1980).

2. T. A. Budd, Mutation analysis: ideas, examples, problems

and prospects. In *Computer Program Testing*, edited B. Chandrasekaran and S. Radicchi, pp. 129–148. North-Holland, New York (1981).
3. M. D. Davis and E. J. Weyuker, A formal notion of program-based test data adequacy. *Inf. and Control* **56**, 52–71 (1983).
4. R. A. DeMillo, R. J. Lipton and F. G. Sayward, Hints on test data selection: help for the practicing programmer. *Computer* **11** (4), 34–41 (1978).
5. W. E. Howden, Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.* **SE-8**, 371–379 (1982).
6. G. J. Myers, *The Art of Software Testing*. Wiley, New York (1979).
7. T. J. Ostrand and E. J. Weyuker, Error-based program testing. *Proc. 1979 Conf. on Information Sciences and Systems*, Baltimore (1979).
8. C. V. Ramamoorthy, S. F. Ho and W. T. Chen, On the automated generation of program test data. *IEEE Trans. Software Eng.* **SE-2**, 293–300 (1976).
9. E. J. Weyuker and T. J. Ostrand, Theories of program testing and the application of revealing subdomains. *IEEE Trans. Software Eng.* **SE-6**, 236–246 (1980).
10. E. J. Weyuker, Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems* **5** (4) (1983).
11. E. J. Weyuker, *Evaluating Software Complexity Measures*. New York University Department of Computer Science Technical Report No. 149 (1985). (To appear in *IEEE Trans. Software Eng.*)
12. E. J. Weyuker, Axiomatizing software test data adequacy. *IEEE Trans. Software Eng.* **SE-12**, 1128–1138 (1986).

APPENDIX
PROOFS OF THEOREMS

Proof of Theorem 1

Clearly, $P(t) = Q(t)$ for all $t \in T$. ■

Proof of Theorem 2

Any branch not exercised could be eliminated using one of the rules without changing the input–output behaviour of the program on the test set T. ■

Proof of Theorem 3

For each $Q \in \Phi(P)$, we know that it is not the case that $Q \equiv P$ (otherwise Φ would not be protected for P), so there is a point t_Q such that $P(t_Q) \neq Q(t_Q)$. Let

$$T = \{t_Q \mid Q \in \Phi(P)\}.$$

Since $\Phi(P)$ is finite, it follows that T is a finite set which is Φ-adequate for P. ■

Proof of Theorem 4

For distinct programs P and Q, we define:

$$\rho(P, Q) = 1 \text{ if } Q \in \Phi(P),$$

$$\rho(P, Q) = 2 \text{ otherwise.}$$

The desired result follows at once. ■

Proof of Theorem 5

A Φ-critical point for a program P is a point such that P differs from some program in Φ(P) only at that point. It therefore follows immediately from the definitions that Φ-critical points for a program P are points which must appear in any Φ-adequate test set. ■

Proof of Theorem 6

Obviously, $\Phi_d(P) \subseteq \Phi_e(P)$. ■

Proof of Theorem 7

Since ρ is stepwise, $\rho(P_i, Q_i) = 1$. Now, for all $t \in T$, we have $Q_i(t) = P(t)$. We know that P is not embedded in P_i; thus P can only be embedded in Q_i if either P always outputs the value b or if $P \equiv Q_i$. But we can eliminate the first possibility because we have assumed that P outputs at least 2 distinct values. Now, if $P \equiv Q_i$, c_i is a d-critical point for P; on the other hand, if P is not equivalent to Q_i, then by definition, $\rho(P, Q_i) > d$. Also,

$$\rho(P, Q_i) \leqslant \rho(P, P_i) + \rho(P_i, Q_i) \leqslant d + 1.$$

Since our distance function is assumed to be discrete, it follows that $\rho(P, Q_i) = d + 1$. ■